正则表达式转换为最小有限确定自动机

本次实验内容主要是为了实现将词法分析中正则表达式转换为最小有限确定自动机的过程。因为这学期课程压力太大了，这个实验最高也只能得2分，所以这个报告我并不打算写的很详细，简单介绍一下我完成整个代码逻辑和构思的过程。

实验代码链接：155chb/regex2MDS: ot1 将正则表达式转化为最小确定有限自动机 (github.com)

正则表达式转为NFA

文法构建

考虑到本次实验中，需要进行识别的文法是正则表达式，它的运算说起来不算复杂。主要包括了

- *——闭包
- +——正则闭包
- ?——可选
- .——任意字符
- ()——作为整体首先进行运算
- |——或运算
- []——选择字符集合中一个字符
- \——转义字符，将后一个符号识别为字符而不是特殊符号（如*+等），或者将\r\t\n作为一个整体识别字符

ps：我一开始就直接这么设计了，写报告的时候发现老师ppt上写到了Lex好像还支持好多骚操作，看着就头疼，不打算弄了。

观察上述几种运算，可以发现以上这些运算主要是**右运算符**，只有或运算和拼接运算属于中间运算符（拼接甚至不需要运算符）。

如果我想要设计一种上下文无关文法并给出其翻译模式，那么很重要的一点就是，当识别到某一个token的时候，如何判断使用哪一条产生式和翻译模式来处理当前的token，为了避免进行回溯导致程序过于复杂，必须设计一种**一条产生式能够唯一对应某一个token的上下文无关文法。**

首先想到的是老师唯一一个讲完的文法——LL(1)文法，

但是使用这种文法有一个很明显的缺陷，LL(1)设计的文法要求消除左递归和左公因子，因此对于*+?这些右运算符时不得不将其放在最左边，这样就会导致其在分析时无法直接找到它的操作对象，造成翻译模式设计起来尤其复杂。

考虑到这一点，仿照LL(1)文法，设计了满足**RR(1)文法**条件的一组产生式，同时为了简化模型，我将[xxxx]直接识别为一个token（其实也不是识别为一个token，只不过处理的时候直接将其作为一个整体处理了，将[]里的一些稍微复杂些的规则过滤掉了，在最一开始将这个工作交给了另一个函数进行处理，这里就相当于将其识别为一个token了，这样说更好理解）。

以下是我设计的文法，从右向左识别字符，

- 0： E0 -> E1 E2
- 1： E1 -> E0 '|'
- 2： E1 -> E0
- 3： E2 -> E3 '*'

- 4： E2 -> E3 '+'
- 5： E2 -> E3 '?'
- 6： E2 -> E3
- 7： E3 -> '(' E0 ')'
- 8： E3 -> '[' N ']'　　这就是上面说的，将[]作为一个整体识别为token（这样的说法更容易让人理解）
- 9： E3 -> '.'
- 10：E3 -> C　　这个C就是表示一个字符，例如字符0，字符a之类的都将其识别为C
- -1： E1 -> ε　　判断使用这条产生式的条件是已经读入了第一个字符了，当前**索引序号已经小于0**了，**没有需要继续读入的字符**了，或者**读到了'('**，这也说明当前不需要继续读入字符了，需要**先将这一步计算完再向前读取字符**

字符串整理

本次实验中，第一个关键的问题就是上面提到的，通过将输入的字符串进行整理以便简化文法的设计。

主要需要进行整理的，其实主要还是**[]**中内容，其它部分该怎么识别就怎么识别就可以了，大家都按照一个规则进行的。

但是，问题就是**[]中识别的规则和[]外是完全不同的**，不仅仅在于其多加了一种运算符-，还在于其转义字符\的作用范围也被改变了，[]外部使用*+?符号需要将其进行转义，而[]内部是不需要考虑这些的。

因此，为了简化文法的设计，就需要将[]的内容转换成一个统一的形式或者将其单独作为一个整体进行处理。

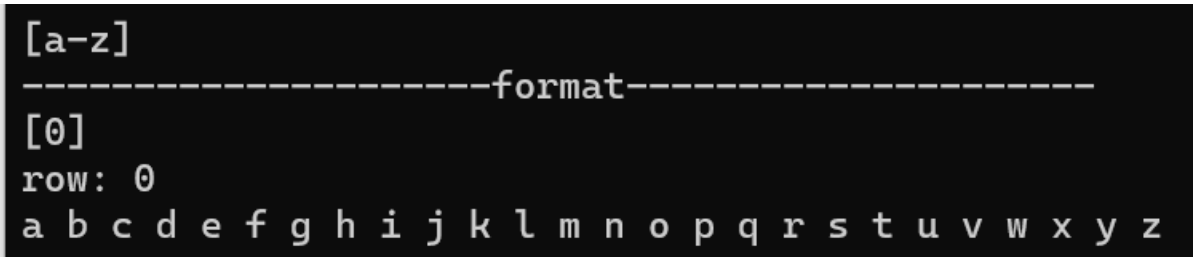
本次实验中，我采取的是一种构思起来比较简单的方式，就是在输入完正则表达式后，将[]内容进行转译，**可以理解为**我**将[abc]用(a|b|c)替换**掉了，这是一种非常直观的实现思路。

但是，由于-运算符的存在，我如果直接进行替换，可能需要反复调整字符串，同时造成多次|操作。而在Thompson构造法中，|操作不得不引入两个新节点，这样的操作很显然造成生成的NFA过于复杂，存在太多的空边，浪费空间。

因此，我选择的方法是将每次读取到[]时，给其分配一个index，让其能索引到一个bool[128]的数组，对于[]所包含的内容，遍历一遍来将其对应的bool数组初始化，每个bool直接对应一个ASCII码。但是还是需要告知构造NFA的函数，应该在这里读取第几个bool数组不是吗？因此，最后将index存进字符串中，也就是说，**假如我遇到一个[abc]，它是我遇到的第一个需要处理的[]，因此我给它分配了index=0，字符串整理之后应该在这个位置存入[0]**，最后构造NFA的函数，就可以在这里识别到一个数字字符串，然后读取到bool数组，**直接创建两个节点（起始节点和终止节点），根据bool数组中存的数据从起始节点连接到终止节点。**

这里代码的逻辑比较乱，直接在报告中展示会比较麻烦，特别占地方，而且感觉并不是十分需要了解代码逻辑，如果感兴趣可以去链接里看一下源码。

以下是字符串整理运行结果截图



NFA数据结构设计

NFA应该使用什么数据结构来保存呢?

根据课上学习到的知识，我当时第一反应是使用有向图来表示，因为NFA说到底就是状态图，那必然是使用图来表示啊。

然后我又考虑了有向图的表示方法，邻接矩阵、邻接链表等等。当时想的是邻接矩阵太占内存了，而且我一开始也不知道应该有多少种状态，我总不能每次新建一个状态就添加一行添加一列吧。所以一开始我是朝着邻接链表的方向进行探索的。

探索了几个小时，我越寻思越感觉不对劲，使用邻接链表表示这个图，那我就必须使用一个数组（或者一个链表）标识状态列表，然后每一个状态有一条属于它的链表，链表里是它所能到达的节点的相关数据（状态序号和输入）。到这里还好，但是再向后细想，如果到了求DFA的时候，我需要求一个闭包，那我就不得不去遍历一遍这个链表才能获取相关信息，即使使用一个由输入来排序的有序链表，它的复杂度依然不可观，而且当时还分析了语法树和一些其它的东西，东西太多烧坏脑子了，导致完全捋不明白各种关系，最后不得不先放弃了这种方式。

静下心之后再进行思考，我很快想到了一个简单的替代方法，那就是将邻接链表再加一维，将输入input从链表节点的结构中薅出来，让它单独作为一个维度存在。这样就可以直接通过数组直接索引到某个状态关于某个input对应的转移状态链表。这样设计虽然造成一些空间浪费，但着实是让人豁然开朗。其实，按照老师课上介绍的内容，这大概就属于**状态转换表**的一个实现，**参数分别是当前状态和输入，输出就是能够转移到的状态链表。**因为输入只有可能是ASCII码对应的字符集合，因此直接使用128作为数组的输入的可能取值范围（其实有大部分输入我根本识别不出来），这样看起来也比较顺眼。

```
6     listNode stateListNFA[STATELISTSIZE_NFA][128];
7     int stateListSizeNFA = 0;
```

这里考虑到一般状态表的状态数不会太多，现在设置的是一个100的上限，如果需要更改修改宏定义就可以了（说实话光改宏定义也不太行，format那一步偷了个懒，不过简单改一下就好，没啥影响）。

数据结构的设计就是这样了，当然为了能够实现最终的目的，还需要给这个变量添加一些函数来系统的管理它

- initStateListNFA()
  这个函数是一个初始化程序，实际就是将stateListNFA设置一些初始值，初始的data都设置为-1，不会指向任何一个状态，因为它们是头节点，头节点不需要它们存数据。此外，因为它是一个全局变量，是在编译时分配内存，因此使用后也不需要释放掉了

```
int initStateListNFA() {
    for (int i = 0; i < STATELISTSIZE_NFA; i++) {
        for (int j = 0; j < 128; j++) {
            stateListNFA[i][j].data = -1;
            stateListNFA[i][j].next = nullptr;
        }
    }
    return 0;
}
```

- freeStateListNFA()

这个函数就是为了程序结束时将listNode的链表数据释放掉，因为他们是运行后malloc的内存，因此需要最后使用free释放掉它们，调用的是一个freeList函数，在listNode.h文件中实现的，使用两个指针将一个链表内存全部释放

```
int freeStateListNFA() {
    for (int i = 0; i < STATELISTSIZE_NFA; i++) {
        for (int j = 0; j < 128; j++) {
            freeList(&stateListNFA[i][j]);
        }
    }
    return 0;
}
```

- newStateNFA()
  这个函数就是为了规范化，里面实现的其实就是返回一个新的系列号（从0开始），然后将状态总数加1

```
int newStateNFA() {
    return stateListSizeNFA++;
}
```

- connectStateNFA(int srcIndex, int dstIndex, int input)
  这个函数就是用来连接两个节点的，实际上就是在状态转移表中对应的链表中插入一个新的节点

```
int connectStateNFA(int srcIndex, int dstIndex, int input) {
    return insertNode(&stateListNFA[srcIndex][input], dstIndex);
}
```

- printfStateNFA()
  这个函数用于输出NFA列表，使用了printfData函数，也是listNode.h文件中定义的，用于输出一个链表中的数据

```
int printfStateNFA() {
    for (int i = 0; i < stateListSizeNFA; i++) {
        for (int j = 0; j < 128; j++) {
            if (stateListNFA[i][j].next != nullptr) {
                printf("state:%5d\tinput:   %c\tnext:    ", i, (char)j);
                printfData(&stateListNFA[i][j]);
                cout << endl;
            }
        }
    }
    return 0;
}
```

生成NFA算法

老师提示说生成NFA的算法，可以使用Yacc工具或者递归下降程序。

然而我首先觉得，使用语法分析器解决词法分析的问题属实是有些奇怪（虽然使用C语言写一个C编译器也很奇怪），这种方式首先被舍弃了。

但是我说实话并不知道老师说的递归下降程序是什么意思，因此我一开始的方向有一些跑偏。一开始打算的是构造一个语法树，然后遍历语法树来完成翻译，弄了一段时间脑子越来越混乱。

最后突然想到可是使用函数调用来完成类似于语法树的翻译过程，因为函数的调用过程也是可以写作一棵函数调用的数的，而按照产生式将翻译的动作写在这些函数中，就相当于按照语法树遍历的顺序执行了这些动作不是吗？这个部分其实老师上课的时候也介绍过，ppt上的语法分析大概就是这么一个逻辑，想到老师的提示，可能这就是一个**递归下降程序**吧。

因此就按照非终结符作为函数的标识，实现一个递归下降程序，这里先**拿E2举一个例子**，

- E2 -> E3 '*' | E3 '+' | E3 '?' | E3

这里就需要**先执行E3的构造函数**，然后**执行对应运算符的动作**，但是想要执行对应的运算符动作，必然**需要知道E3执行得到的初始节点和终止节点。**

例如，***执行的操作就是从起始节点指向终止节点，从终止节点指向起始节点**（这里是想节省一下，尽量减少状态的总数，不知道这样改会不会有什么bug，不过测试的时候没有发现这里有什么问题，如果发现了bug简单修改一下就行了，因为这里的逻辑还是很清晰的）。

因此，**使用一个参数向下传递，传递的是索引或者指针，使得E2能将获得E3起始节点的索引号和终止节点的索引号**，如下

```
int do_2(int& startStateIndex, int& endStateIndex) {
    int productionIndex = getProductionIndex(2);
    if (productionIndex == 3) {
        // 对于产生式3，需要将E3的结束状态连接到E3的开始状态
        // 并将E3的开始状态连接到E3的结束状态
        // （改了一下，不是上课讲的那种了）
        endCharIndex--;
        do_3(startStateIndex, endStateIndex);
        connectStateNFA(startStateIndex, endStateIndex, '\0');
        connectStateNFA(endStateIndex, startStateIndex, '\0');
    }
    else if (productionIndex == 4) {
        // 对于产生式4，需要将E3的结束状态连接到E3的开始状态
        endCharIndex--;
        do_3(startStateIndex, endStateIndex);
        connectStateNFA(endStateIndex, startStateIndex, '\0');
    }
    else if (productionIndex == 5) {
        // 对于产生式5，需要将E3的开始状态连接到E3的结束状态
        endCharIndex--;
        do_3(startStateIndex, endStateIndex);
        connectStateNFA(startStateIndex, endStateIndex, '\0');
    }
    else if (productionIndex == 6) {
        do_3(startStateIndex, endStateIndex);
    }
    return 0;
}
```

同时还需要注意，如果当前识别到的是运算符，一定是**要先将运算符读取出来，将剩余的部分用于E3的识别和分析**，因此就**使用endCharIndex指示当前读取到了哪个字符，因为是从后向前读取的，因此就将其自减1。**

其它的do函数，其主体结构其实也都大差不差。但都还是有一些需要注意的部分，以下我将简单介绍一下，

do_1函数中，由于无论是拼接还是或操作，都需要两个对象，E1的生成式很显然都只包含了一个E0，它的相关运算的右侧操作数实际上要到E0的生成式中才能找到。因此，很显然**拼接和或操作并不是在do_1函数中实现的**，**do_1函数只能实现读入'|'操作和递归调用do_0函数**，而实际的**拼接和或操作是在它的上一层也就是do_0函数中实现的**，但这样就**必须让do_0函数能知道do_1对应的产生式是什么**，因此就将这两个函数设计成了这样（我觉得注释已经写的很清晰了），

```
int do_1(int& startStateIndex, int& endStateIndex) {
    // 在E1中，不需要做任何的连接操作
    // 只需要注意一下，当执行产生式2时将字符的位置向前移动一位
    int productionIndex = getProductionIndex(1);
    if (productionIndex == -1)
        return -1;
    if (productionIndex == 1)
        endCharIndex--;
    do_0(startStateIndex, endStateIndex);
    return productionIndex;
}
```

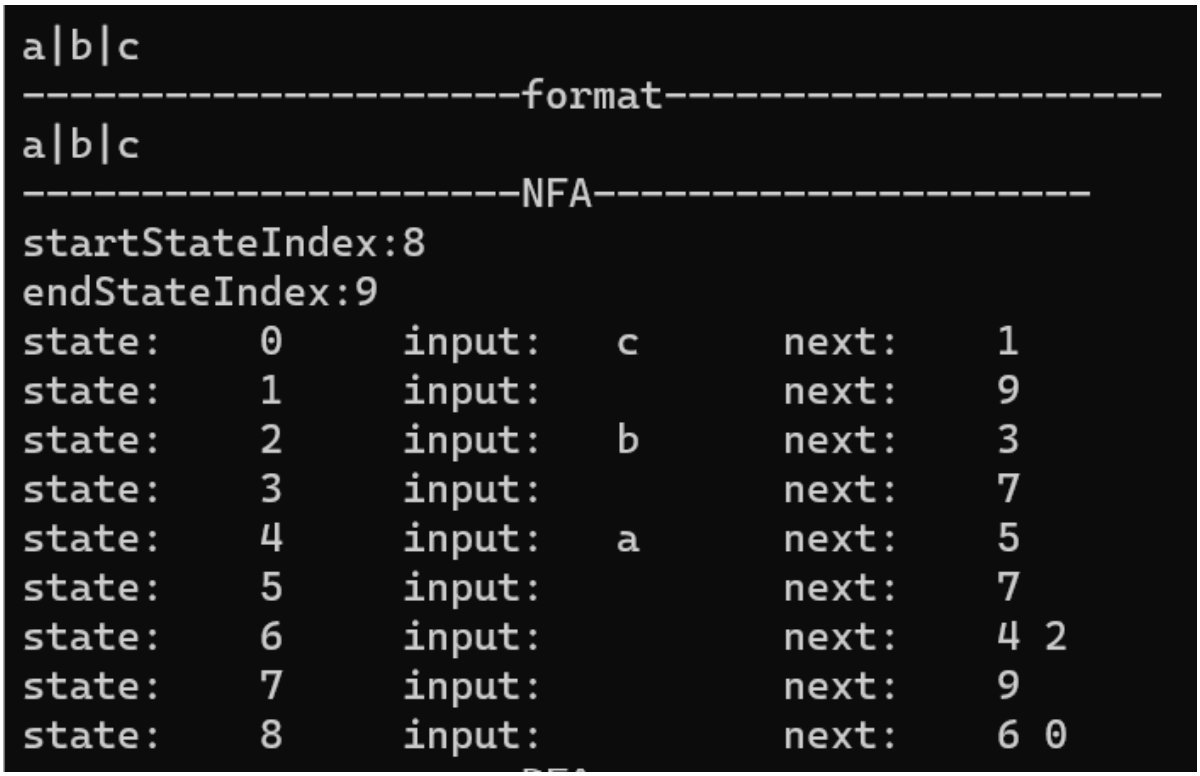
```
int do_0(int& startStateIndex, int& endStateIndex) {
    // 在E0中，需要获取到E1的产生式，根据产生式的不同，做不同的连接操作
    int startIndex2, endIndex2, startIndex1, endIndex1;
    do_2(startIndex2, endIndex2);
    int productionIndex1 = do_1(startIndex1, endIndex1);
    if (productionIndex1 == -1) {
        // 产生式为-1时，说明E1为空，直接将E0的开始状态和结束状态设置为E2的开始状态和结束状态
        startStateIndex = startIndex2;
        endStateIndex = endIndex2;
    }
    else if (productionIndex1 == 1) {
        // 产生式为1时，说明E1为E0 |，新建两个节点，然后连接
        // 这里之前自以为是，改了一下，果然出问题了，人麻了，所以改回了课上讲的方法
        startStateIndex = newStateNFA();
        endStateIndex = newStateNFA();
        connectStateNFA(startStateIndex, startIndex1, '\0');
        connectStateNFA(startStateIndex, startIndex2, '\0');
        connectStateNFA(endIndex1, endStateIndex, '\0');
        connectStateNFA(endIndex2, endStateIndex, '\0');
    }
    else if (productionIndex1 == 2) {
        // 产生式为2时，说明E1为E0，需要将E1和E2拼接起来
        startStateIndex = startIndex1;
        endStateIndex = endIndex2;
        connectStateNFA(endIndex1, startIndex2, '\0');
    }
    return 0;
}
```

do_3函数其实就是执行了一下创建最小节点的过程，不过还是**需要注意一下()和[]的实现。')'标识了使用的产生式，而'('则标识了一个整体的结束，因此在识别E1 -> ε也需要考虑到这种情况。而[]则是直接查bool数组直接连接节点，.也是同理**（将除了\0以外的都连接上了，即使有些字符其实无法读取出来，不过也无所谓了，懒得考虑太多了）。

```
int do_3(int& startStateIndex, int& endStateIndex) {
    int productionIndex = getProductionIndex(3);
    if (productionIndex == 7) {
        endCharIndex--;
        do_0(startStateIndex, endStateIndex);
        endCharIndex--;
    }
    else if (productionIndex == 8) {
        startStateIndex = newStateNFA();
        endStateIndex = newStateNFA();
        endCharIndex--;
        int index = regex[endCharIndex--] - '0';
        if (regex[endCharIndex] != '[')
            index += 10, endCharIndex--;
        for (int i = 0; i < 128; i++)
            if (M[index][i])
                connectStateNFA(startStateIndex, endStateIndex, (char)i);
        endCharIndex--;
    }
    else if (productionIndex == 9) {
        startStateIndex = newStateNFA();
        endStateIndex = newStateNFA();
        for (int i = 1; i < 128; i++)
            connectStateNFA(startStateIndex, endStateIndex, (char)i);
        endCharIndex--;
    }
    else if (productionIndex == 10) {
        startStateIndex = newStateNFA();
        endStateIndex = newStateNFA();
        connectStateNFA(startStateIndex, endStateIndex, regex[endCharIndex--]);
        if (regex[endCharIndex] == '\\')
            endCharIndex--;
    }
    return 0;
}
```

其实对比一下能发现实现或操作和[]操作或者.操作时节点连接是不一样的，比如对于一个**[abc]它等价于a|b|c**，但是观察一下这两个式子得到的NFA能看出它们的**构造过程实际上是有一些区别的**，

```
[abc]
----------------------format----------------------
[0]
row: 0
a b c
----------------------NFA-----------------------
startStateIndex:0
endStateIndex:1
state:    0     input:   a      next:    1
state:    0     input:   b      next:    1
state:    0     input:   c      next:    1
```



差别还是挺明显的吧，可以看到**[]直接在两个状态上插入输入对应的边，而|则需要新建开始状态和终止状态，然后分别连接首尾。**这样设计主要是因为|需要考虑到前向的边，**例如a+|b，它如果直接将a+和b的开始状态和终止状态拼接的话，由于a+的终止状态是可以指向开始状态的，就会导致b实际也能被遍历到许多次，就完全改变了状态机的含义。**这里起始修改*和+的运算时执行的操作也能修改这个bug，但当时觉得改一个或会更容易，所以就直接只改了一个或操作。

其它的就没什么特别需要注意的地方了，获取产生式序号的**getProductionIndex函数的逻辑实际非常简单，就是类似实现了一个预测分析表，只不过是使用switch进行判断的**，具体读取到什么输入对应一个什么产生式，看设计的文法就好，特别的直观。

转换结果截图

本次实验报告中，测试程序就使用的是老师上课ppt中使用到的一个正则表达式（NFA例2）(a(b*c))|(a(b|c+)?)，测试结果截图如下

```
(a(b\*c))|(a(b|c+)?)
--------------------format--------------------
(a(b\*c))|(a(b|c+)?)
--------------------NFA--------------------
startStateIndex:16
endStateIndex:17
state:    0     input:    c      next:    1
state:    1     input:           next:    0 5
state:    2     input:    b      next:    3
state:    3     input:           next:    5
state:    4     input:           next:    2 0 5
state:    5     input:           next:    17
state:    6     input:    a      next:    7
state:    7     input:           next:    4
state:    8     input:    c      next:    9
state:    9     input:           next:    17
state:   10     input:    *      next:    11
state:   11     input:           next:    8
state:   12     input:    b      next:    13
state:   13     input:           next:    10
state:   14     input:    a      next:    15
state:   15     input:           next:    12
state:   16     input:           next:    14 6
```

感兴趣的话可以画一下看看这个图对不对，我懒得在实验报告中再画一个图了，或者直接看最后MDS的结果，那个最容易看对不对。

NFA转为DFA

DFA数据结构设计

类似于NFA，DFA为了能有一个清晰的表示，实际上使用的也是一个**状态转移表**，只不过不像NFA需要在一个表项中写入多个数据，因为DFA是一个唯一确定的转移目标，因此**直接使用int类型保存就可以了**，

```
int stateListDFA[STATELISTSIZE_DFA][128];
int stateListSizeDFA = 0;
```

不过需要注意的是，和NFA不同的是，**DFA的终止状态可能不止一个**，这是设计的链表结构就又可以派上用场了，

```
listNode endStateIndexListHeadDFA;
```

同时，因为求解DFA的过程中，**需要记录某一个DFA状态对应的是哪几个NFA状态的集合**，因此又设计了一个链表数组结构用于存放一个DFA状态对应的NFA状态集合。

```
listNode DFA2NFASetHead[STATELISTSIZE_DFA];
```

除此之外，DFA也有对应的初始化、输出函数、连接函数等等，这里逻辑和NFA大体是类似的，因此不再赘述，感兴趣看看源码。

生成DFA算法

如何计算将一个从一个NFA转化为DFA呢，老师上课介绍过具体的方案，那就是求ε闭包和求δ，

1. 将NFA的开始状态的ε闭包作为第一个状态
2. 然后不断计算δ和求ε闭包得到一个新的状态，检查其是否和过去已经添加的状态重合，不重合就当作一个新状态添加到DFA中，然后连接两个状态
3. 遍历过所有已存在状态后，检查一遍所有状态，如果包含NFA中的终止状态，就将其设置为DFA的终止状态

我所实现的生成DFA算法实际上也是基于这个过程进行设计的

那首先需要解决的问题就是求ε闭包和求δ集了，

计算ε闭包

求epsilon闭包的过程其实也不算很难实现，**使用一个队列，先来的先处理，计算得到这个状态走空输入能到达的状态，将其添加到队列中，注意不要让处理过的状态在添加到队列的尾部就好了。**

至于队列这个数据结构，正好还是使用**listNode来完成，使用链表一步一步向后移动，同时将新节点向链表尾部**，正好我一开始设计这个链表结构的时候就没考虑将其做成一个有序的，都是向链表尾部进行插入，**插入的时候还正好检查了是否该链表节点已经存在。**

```
int epsilonClosure(listNode* dstHead, listNode* srcHead) {
    listNode* curDst, * curSrc;
    // 使用一个临时的头节点，避免dstHead和srcHead相同时的错误
    listNode tempHead;
    tempHead.data = -1;
    tempHead.next = nullptr;
    curSrc = srcHead->next;
    curDst = &tempHead;
    while (curSrc != nullptr) {
        // 对于每个src中的输入，将其插入到tempHead中
        insertNode(&tempHead, curSrc->data);
        while (curDst->next != nullptr) {
            // 将tempHead中的每个数据的epsilon闭包插入到tempHead中
            curDst = curDst->next;
            insertSet(&tempHead, curDst->data, '\0');
        }
        curSrc = curSrc->next;
    }
    // 如果src和dst是同一个节点，需要先释放掉内存然后在把指针转移
    if (dstHead == srcHead)
        freeList(dstHead);
    dstHead->next = tempHead.next;
    return 0;
}
```

其中使用的insertSet函数就是将一个NFA中的状态某个输入对应的链表全部添加，逻辑比较简单，也不算重点内容，因此不在这里多说明了。

这里的逻辑我是这么写的，可能看起来比较乱，其实先将srcHead中的链表节点都添加到tempHead后，再遍历链表添加ε输入到达状态应该也是可以的。不过基于代码没bug就不改的原则，我也懒得研究改好还是不改好了。

计算δ集合

计算δ其实逻辑比求ε闭包还要简单，因此我也不太想多说啥了，看看代码就好了

```
int delta(listNode* dstHead, listNode* srcHead, char input) {
    listNode* cur= srcHead->next;
    // 类似于上一个函数，虽然正常程序中应该不存在dst等于src的情况
    listNode tempHead;
    tempHead.data = -1;
    tempHead.next = nullptr;
    while (cur != nullptr) {
        insertSet(&tempHead, cur->data, input);
        cur = cur->next;
    }
    if (dstHead == srcHead)
        freeList(dstHead);
    dstHead->next = tempHead.next;
    return 0;
}
```

计算NFA算法实现

这里先看代码，

```
int createDFA(int startStateIndexNFA, int endStateIndexNFA) {
    // 对初始节点进行一个初始化，当前唯一元素是NFA的初始节点的闭包
    int startStateIndexDFA = newStateDFA();
    insertNode(&DFA2NFASetHead[startStateIndexDFA], startStateIndexNFA);
    epsilonClosure(&DFA2NFASetHead[startStateIndexDFA], &DFA2NFASetHead[startStateIndexDFA]);
    int curIndexDFA = startStateIndexDFA;
    // 定义一个临时的头，用于接收一个新的闭包集合
    listNode tempHead;
    tempHead.data = -1;
    tempHead.next = nullptr;
    while (curIndexDFA < getStateListSizeDFA()) {
        for (int i = 1; i < 128; i++) {
            // 遍历每一个非空字符
            tempHead.next = nullptr;
            delta(&tempHead, &DFA2NFASetHead[curIndexDFA], i);
            epsilonClosure(&tempHead, &tempHead);
            if (tempHead.next == nullptr)
                continue;   // 如果闭包为空，跳过
            // 检查是否已经存在这个闭包集合
            int flag = 0, j = 0;
            for (; j < getStateListSizeDFA(); j++) {
                flag = listCmp(&DFA2NFASetHead[j], &tempHead);
                if (flag == 1)
                    break;  // 发现存在就直接跳出循环
            }
            if (flag == 0) {
                // 如果不存在相同的，就新建一个节点，把指针转移过去
                j = newStateDFA();
                DFA2NFASetHead[j].next = tempHead.next;
            }
            else {
                // 如果存在相同的，就释放掉获取到的新的闭包集合，它不需要保存了
                freeList(&tempHead);
            }
            connectStateDFA(curIndexDFA, j, i);
        }
        curIndexDFA++;
    }
    // 最后遍历一遍结果，把所有的终态都加入到终态列表中
    for (int i = 0; i < getStateListSizeDFA(); i++) {
        if (searchData(&DFA2NFASetHead[i], endStateIndexNFA) == 1)
            insertNode(&endStateIndexListHeadDFA, i);
    }
    return 0;
}
```

懒得细扣了，简单介绍一下代码的逻辑和结构吧

- 首先将开始节点的ε闭包作为第一个状态
- 定义一个tempHead，用于存放δ(id, input)和其ε闭包
- 遍历每一个添加进DFA的状态，对于每一个input求一次δ和ε闭包
    - 如果得到的结果是一个空集就跳过后面的步骤，继续查下一个input
    - 否则，遍历一遍是否是和之前的某一个状态重合了，如果有，j的数值应该就等于那个状态。
    - 检查flag标记，如果有相同的就把查到的释放掉，如果没有就要新申请一个状态将链表转接过去。
    - 连接两个状态，从当前节点到j标识的那个节点，input就是i
- 最后再遍历一遍，把所有的终止状态找出来添加到终态集合中

这里我使用了函数listCmp和searchData，这两个函数也在listNode.h中，

listCmp的功能就是比较两个list是否相同（这里的相同指的是内容相同，和顺序无关，其实就是先计算一下两个链表的长度，长度不等就返回0，长度相等就不断调用searchData查看两个链表的内容是否相同，如果完全相同才返回1）。

searchData的功能就是检查一个链表中是否包含某个数据，遍历一遍链表，有对应数据就返回1，否则返回0

DFA转MDS

MDS的数据结构和DFA完全相同，因此不多说了，不过因为要算法中要反复查看DFA的某个状态对应到了MDS的哪个状态，因此我设计了一个

```
int DFA2MDS[STATELISTSIZE_DFA];
```

它是从DFA index获取到MDS index，因为**算法中需要不断求DFA对应的MDS**，如果从MDS那里进行设计，就可能需要遍历所有MDS，因此还是从DFA到MDS的设计会比较合理。

生成MDS算法实现

MDS一个很重要的部分就是输入能否区分两个状态。

不过好在我将DFA中所有输入无法到达任何节点的数值设置为了-1，那其实就可以将-1当作一个死状态，死状态不会去求它的输入会不会到达某个节点，同时死状态唯一存在。

如果两个状态对于某个输入所能到达的节点都是完全相同的（包括到达死状态），那其实就说明对于任何输入都是无法区分这两个状态的，因此只需要对比一下stateListDFA中的数值，就能知道这两个状态是否可区分了。

写到这里的时候，真的一点也写不动了，因此还是直接看代码吧，

```
int initDFA2MDS() {
    int stateListSizeDFA = getStateListSizeDFA();
    bool flag = 1;  // 定义一个标记变量，1表示该状态与之前的状态不相同
    for (int i = 0; i < stateListSizeDFA; i++) {
        flag = 1;
        for (int j = 0; j < i; j++) {
            for (int k = 0; k < 128; k++) {
                // 把所有可能输入遍历一遍，如果均相等则不可区分
                flag = stateListDFA[i][k] != stateListDFA[j][k];
                if (flag)   // 如果发现可区分，就去退出循环检查下一个
                    break;
            }
            if (!flag && (searchData(&endStateIndexListHeadDFA, i) == searchData(&endStateIndexListHeadDFA, j))) {
                // 如果找到了不可区分的就可以标记并退出循环了
                // 这里再判断一下它们是不是都属于非终态或都属于终态
                DFA2MDS[i] = DFA2MDS[j];
                break;
            }
        }
        if (flag)   // 如果没有找到不可区分的，就让它申请一个新的MDS
            DFA2MDS[i] = newStateMDS();
    }
    return 0;
}
```

使用这个函数，就可以实现将DFA状态全部找到对应的MDS状态，这是其实MDS的节点也都分配了出来，节点的数量已经确定了，只需要在后续将边都添加上就可以了。

这个函数中flag变量的使用可能看起来逻辑不是很顺畅，不过我感觉注释已经尽可能讲的清晰了。

分配好节点之后，所需要做的事情也比较简单了，对DFA中每一个可达的边，都创建一条MDS中的边，最后还是把终止状态处理，

```
int createMDS() {
    initDFA2MDS();
    for (int i = 0; i < getStateListSizeDFA(); i++) {
        for (int j = 0; j < 128; j++) {
            if (stateListDFA[i][j] != -1)
                connectStateMDS(DFA2MDS[i], DFA2MDS[stateListDFA[i][j]], j);
        }
    }
    listNode* cur = endStateIndexListHeadDFA.next;
    while (cur != nullptr) {
        insertNode(&endStateIndexListHeadMDS, DFA2MDS[cur->data]);
        cur = cur->next;
    }
    return 0;
}
```

至此，就完成了从正则表达式到NFA到DFA到MDS的全过程了。